

Change Impact Analysis for AspectJ Programs

Sai Zhang, Zhongxian Gu, Yu Lin, Jianjun Zhao
School of Software
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
saizhang, ausgoo, linyu1986, zhao-jj@sjtu.edu.cn

ABSTRACT

Change impact analysis is a useful technique for software evolution. It determines the effects of a source editing session and provides valuable feedbacks to the programmers for making correct decisions. Recently, many techniques have been proposed to support change impact analysis of procedural or object-oriented software, but seldom effort has been made for aspect-oriented software. In this paper we propose a new change impact analysis technique for AspectJ programs. At the core of our approach is the *atomic change* representation which can precisely capture semantic differences between two versions of an AspectJ program. We also present a change impact model, based on static AspectJ call graph construction, to determine the impacted program parts, affected tests and their responsible affecting changes. As an application of change impact analysis, we discuss how our model can help programmers locate the exact failure reason by narrowing down those affecting changes when debugging AspectJ programs.

The proposed techniques have been implemented in Celadon, a change impact analysis framework for AspectJ programs. We performed an experimental evaluation of the proposed techniques on 24 versions of 8 AspectJ benchmarks. The results show that our proposed technique can effectively perform change impact analysis and provide valuable debugging information for AspectJ programs.

1. INTRODUCTION

During software evolution, software changes is an essential operation that introduces new functionalities or fixes bugs in the existing system, or modifies the formal implementation if the requirements were not correctly addressed. Usually after a long session of code editing, the nontrivial combination of these small changes may affect other parts of program unexpectedly, such as the non-local changes in object-oriented languages due to the extensive use of sub-typing and dynamic dispatch. Those likely ripple-effects of software changes may indicate potential defects in the updated version. When regression testing fails, it may be difficult for programmers to locate the culprit code by searching through the source. Moreover, when programmers developing regression test drivers over time to confirm the added or modified functionality of software, it is also difficult to have a clear view of whether the existing test drivers are adequate to cover all affected parts in order to keep the new version working properly with respect to previous releases.

Software change impact analysis [6] is a technique that assesses which parts of a program can be affected if a proposed change is made for the program. It can be used to predict the potential impact of the changes before they applied, or to estimate the potential side-effect of changes. The information provided by change impact analysis would be very important for developers to make cor-

rect decisions and then to eliminate potential risks during software evolution.

Aspect-Oriented Programming (AOP) [12] has been proposed as a technique for improving separation of concerns in software design and implementation. In an aspect-oriented system, the basic program unit is an aspect rather than a procedure or a class. An aspect with its encapsulation of state with associated advice is a significant different abstraction than the procedure unit within procedural programs or the class unit within object-oriented programs. An AspectJ program can be divided into two parts: *base code* which includes classes, interfaces, and other language constructs as in Java, and *aspect code* which includes aspects for modeling crosscutting concerns in the program. With the inclusion of join points, an aspect woven into the base code is solely responsible for a particular crosscutting concern, which raises the system's modularity among aspects and classes. However, when implementing changes in aspect-oriented programs, it involves more complex impact than in the traditional programming languages:

- The woven aspect may change control and data dependency to the base code, and changing the aspect code can significantly affect the semantics of base code.
- Because of the coupling introduced between the base code and aspect code, when either changes in the base code (like renaming classes, fields and methods) or the aspect code (like adding a new pointcut or advice) occur, the semantics of pointcuts may be silently altered and no compiler or weaver can guarantee the correctness.
- Failures of regression testing could either result from changes in the base code or a particular aspect. The more complex cases raise from the interactions between the base and aspect code or even the aspect weaving sequences. In such a case, no single location corresponds to the failure, and the difficulty of finding failure-inducing changes rises dramatically.

Although many change impact analysis techniques have been presented in the literature, most of the work has been focused on procedural or object-oriented software [5, 6, 13, 15, 16]. Since the executable code of an AspectJ program is pure Java bytecode, an obviously approach is to apply directly the existing change impact analysis techniques for Java to the bytecode. In fact, this requires an analysis to build and maintain a map that associate the effects of each element in the bytecode back to those of its corresponding element in the source code. However, as pointed out by Xu's experiment study in [10], there is a significant discrepancy between the AspectJ source code and the woven Java bytecode. This makes it extremely hard to establish such an association map. For example, the *cflow* pointcut requires code to be woven at many places in the base program, both at the so-called "update" shadows and

the "query" shadows [11]. Moreover, the mapping relationship between source-level and byte-code level is specific to the AspectJ compiler being used; different compilers or even different versions of the same compiler can create completely different mappings.

An alternative approach, which taken in this paper, is to perform change impact analysis on the source code of AspectJ programs. However, source code level analysis is complicated by the semantic complexity of the AspectJ programming language, such as the pointcut types, multiple advice invoked at the same join point, the existence advice precedence et al. Therefore, an appropriate change impact analysis technique which can capture the subtle semantic differences and its effects in AspectJ programs are needed. In this paper, we proposed a new change impact analysis technique for AspectJ programs. Our technique uses static AspectJ call graph [14] as the basis of analysis. At the core of our approach is the *atomic change* representation, which can capture precisely the semantic differences between two AspectJ program versions. Based on call graph construction and atomic change representation, the impacted program parts, affected tests, and the responsible change for each affected tests can be effectively identified.

To investigate the feasibility and effectiveness of our proposed technique, we have implemented Celadon, a change impact analysis framework for AspectJ programs, and Flota [23], a fault localization tool for AspectJ programs as an application of the change impact technique. An experimental study of 24 versions of 8 AspectJ benchmarks was performed and the result shows that our proposed technique effectively provides tool supports in impact analysis and valuable debugging support for AspectJ programs. The main contributions of this paper are:

- Proposal of a new change impact analysis technique for AspectJ programs, with a catalog of *atomic changes* representations and their inter-relationships.
- Elaboration of a novel change impact analysis model for AspectJ programs, which can capture the semantic changes of the programs precisely and determine the affect program parts, affected tests and their responsible changes effectively.
- Representation of the debugging support for AspectJ programs as an application of the change impact model, which can help programmers to debug AspectJ programs by narrowing down the likely failure-inducing changes.
- Implementation of Celadon, a change impact analysis framework for AspectJ programs and Flota, a fault localization tool based on the proposed techniques.
- Experimental evaluation of the proposed change impact technique. The results show that the accuracy and cost of our approach is a promising solution to analyze the change impacts of AspectJ programs.

The rest of this paper is organized as follows. Section 2 introduces a motivating example that gives intuition about our approach. Section 3 presents a category of atomic changes for aspect-related constructs, as well as the inter-relationships between these atomic changes. Section 4 presents a change impact analysis model for AspectJ programs. In Section 5, we present the debugging support for aspect-oriented programs as one of the applications of our model. In Section 6, we discuss some technical issues of the implementation of Celadon. Section 7 describes an empirical evaluation of our model, using the Celadon framework. Related work and concluding remarks are given in Section 8 and Section 9, respectively.

2. MOTIVATING EXAMPLE

We next present a simple example to give an informal overview of our change impact technique.

Figure 1 shows an original Java program version which consists of five classes A, B1, B2, C and Tests. Class Tests contains four JUnit [4] tests, Tests.test1, Tests.test2, Tests.test3, Tests.test4, which serve to test all the methods in class A, B1, B2 and C.

```
class A {
    public int i, j;
    public void m() { i++; }
    public void n() { j++; }
}
class B1 extends A { }
class B2 extends A { }
class C extends B1 { }
class Tests extends TestCase {
    public void test1 {
        A a = new A(); a.m(); a.n();
        Assert.assertTrue(a.i == a.j);
    }
    public void test2 {
        A b1 = new B1(); b1.m(); b1.n();
        Assert.assertTrue(a.i == a.j);
    }
    public void test3 {
        A b2 = new B2(); b2.m(); b2.n();
        Assert.assertTrue(a.i == a.j);
    }
    public void test4 {
        A c = new C(); c.m(); c.n();
        Assert.assertTrue(a.i == a.j);
    }
}
```

Figure 1: Original version of example program with test cases

In order to show the impact of changes, we assume a sequence of source modifications in Figures 2, 3, 4 and 5 respectively. The aspects in these changes are newly introduced and the editing part is marked by underline. The first change in Figure 2 is adding a new aspect IntroduceM2B which introduces a method m() to override the existing method of classes B1 and B2. This change can affect the method dispatch of the original program. The InterceptAspect aspect introduced in the same Figure 2 intercepts the calling procedure of each method in class A and increases the variables before the method execution. In Figure 3, the aspect IntroduceField declares two inter-typed fields countM and countN for class A. Associated with this declaration, class C modifies its methods m() and n() to increase the variable countM and countN each time when they are called. The third program modification showed in Figure 4 is to add a new method p() to class A. Although this new method has not been called directly by any existing method, it accidentally matches the pointcut defined in Figure 2. This situation should also be taken into consideration when analyzing the change impact. The final program change is related to pointcut modification which is a frequent action in the development of aspect-oriented software. As showed in Figure 5, pointcut callPoints() has been modified and after this change, it can only match when A.p() is called.

```
aspect IntroduceM2B {
    public void B1.m() { i = i + 2; }
    public void B2.m() { i = i + 1; }
}
aspect InterceptCall {
    pointcut callPoints(A a) : call(* A.*(..))&&this(a);
    before(A a):callPoints(a) { a.i ++; a.j ++; }
}
```

Figure 2: The introduced aspects

After applying these changes to the original Java program in Figure 1, we can get a new released version. Given these two versions of the program together with a set of test cases, we first perform a

```

aspect IntroduceField {
    public int A.countM = 0;
    public int A.countN = 0;
}
class C extends B1 {
    public void m() {countM++;}
    public void n() {countN++;}
}

```

Figure 3: The introduced aspect and source edit in class C

```

class A {
    public void m() { i++; }
    public void n() { j++; }
    public void p() { j = j+2; }
}

```

Figure 4: Source code editing in class A

static analysis on the source code to first compute the set of atomic changes which represent the semantic differences between two versions. Then we determine the set of *propagation changes*¹, which can be used to identify the affected program fragments and the affected test cases in a coarse method level based on the dependence relationship in call graph. In the third step, the responsible affecting changes for each affected test cases are identified.

Consider only the changes in Figure 3, the changes in method `C.m()` corresponds to an atomic change **CM** (*Changed Method*), and the introduction of field `A.countM = 0` is decomposed into two atomic changes **INF** (*Introduce New Field*) and **CIFI** (*Change an Introduction Field Initializer*). Note that atomic change **CM** is syntactically dependent on **INF** because changing method body would lead to an invalid program unless field `A.countM` is introduced. In these atomic changes, **CM** corresponds to a node in the call graph of `Tests.test4` whose behavior maybe affected. Therefore, we compute the affected test cases is `Tests.test4` and its affecting changes is **CM**.

3. ATOMIC CHANGES

We next present a catalog of atomic changes for AspectJ programs. These atomic changes focus on the aspect-related constructs and represent the source code modifications at a coarse-grained model (that is, method-level). Our change impact model relies on the computation of atomic changes. We assume that the original and the updated programs to be both syntactically correct and compilable, and any source code changes can be captured by a set of *atomic changes* that is amenable to analysis. Since AspectJ is a seamless extension to Java, the atomic changes identified for Java in [16] should also be used to model the change impact of AspectJ programs.

3.1 Atomic Changes for AspectJ

Table 1 shows a category of atomic changes for AspectJ programs we defined. Most of these changes are self-explanatory except for **AIC**, which will be explained later in detail. We ignore several types of source code changes that have no direct semantic impact on the program behavior. These include changes to access right of aspects, pointcuts, inter-type declarations, and the addition or deletion of a **declare error** statement which only affects compile-time behavior.

3.1.1 Aspect, Pointcut, and Advice Changes

¹We define *propagation change* to be the change that will alter the behavior of other part of program. Changes like *define an unused variable* is not considered to be *propagation change*.

```

aspect InterceptAspect {
    pointcut callPoints(A a) : call(* A.p(..)) && this(a);
    before(A a):callPoints(a) { a.i++; a.j++; }
}

```

Figure 5: Pointcut editing in aspect InterceptAspect

Abbreviation	Atomic Change Name
AA	Add an Empty Aspect
DA	Delete an Empty Aspect
INF	Introduce a New Field
DIF	Delete an Introduced Field
CIFI	Change an Introduced Field Initializer
INM	Introduce a New Method
DIM	Delete an Introduced Method
CIMB	Change an Introduced Method Body
AEA	Add an Empty Advice
DEA	Delete an Empty Advice
CAB	Change an Advice Body
ANP	Add a New Pointcut
CPB	Change a Pointcut Body
DPC	Delete a Pointcut
AHD	Add a Hierarchy Declaration
DHD	Delete a Hierarchy Declaration
AAP	Add an Aspect Precedence
DAP	Delete an Aspect Precedence
ASED	Add a Soften Exception Declaration
DSED	Delete a Soften Exception Declaration
AIC	Advice Invocation Change

Table 1: A catalog of atomic changes in AspectJ

Seven atomic changes in Table 1 correspond to the changes of aspect, pointcut and advice constructs. **AA** and **DA** denote the set of adding and deleting an empty aspect, respectively. Similarly, **ANP** and **DPC** denote the adding and deleting of a pointcut, while **CPB** denotes the change of a the pointcut body. **AEA**, **DEA** and **CAB** capture the changes of adding, deleting and modifying advice body, respectively. Note that adding an advice, for example the `before():callPoints()` in aspect `InterceptAspect` in Figure 2, is decomposed into three steps: the addition of pointcut `callPoints` (**ANP** atomic change), the addition of an empty advice `before():callPoints()` (**AEA** atomic change), and the insertion of advice body (**CAB** atomic change). Each step is mapped to exactly one atomic change.

3.1.2 Inter-Type Declaration Changes

Six atomic changes in Table 1 represent the changes caused by inter-type declaration mechanism in AspectJ. **INF**, **DIF** and **CIFI** denote the set of adding, deleting and changing of an inter-type field. **INM**, **DIM** and **CIMB** capture changes of introducing a new method, deleting and changing an existing method, respectively. For example, the introduction of field `A.countM` in Figure 3 is decomposed into two steps: introducing a new field and changing the introduced field initializer, which correspond to atomic change **INF** and **CIFI**, respectively.

3.1.3 Hierarchy and Aspect Precedence Changes

Changes to the class hierarchy or the aspect weaving precedence also have significant effects on the program behavior. We use two atomic changes **AHD** and **DHD** to denote the action of adding and deleting a hierarchy declaration in AspectJ. Other two atomic changes **AAP** and **DAP** are used to capture the aspect precedence changes. For simplicity, in our approach, any changes to the hierarchy declaration (such as the **declare parent** statement) or aspect precedence declaration will be transformed into two steps: deleting the existing declaration and adding a new one.

3.1.4 Soften Exception Changes

AspectJ provides a special mechanism to specify that a particular kind of exception, if thrown at a join point, should bypass Java's usual static exception checking system and instead be thrown as an `org.aspectj.lang.SoftException`. We define two atomic changes **ASED** and **DSED** to capture the adding and deleting soften exception declarations. Any change to the soften exception declaration statement is decomposed into two steps: deleting the original declaration and adding a new one.

3.1.5 Advice Invocation Changes

Changes to the base or aspect code may cause lost or additional matches of join points, which may result in accidental advice invocations. Therefore, we define the atomic change **AIC** to represent the advice invocation change. The **AIC** change reflects the semantic differences between an original version P and an edited program version P' in the form of a set of tuples $\langle \text{joinpoint}, \text{advice} \rangle$, which indicates that the advice invoking at the above join point has been changed. Since neither join point nor advice is named construct, we assign name signatures to them to solve this engineering issue in implementation. The formal definition of **AIC** is showed as follows:

$$\mathbf{AIC} = \{ \langle j, a \rangle \mid \langle j, a \rangle \in ((J' \times A' - J \times A) \cup (J \times A - J' \times A')) \}$$

where J and A are the sets of join points and advices in the original program, and J' and A' are the sets of join point and advices in the modified program. $J \times A$ denotes the matched join points and advice tuple set in the original program while $J' \times A'$ denotes the matched tuple set in the updated program version.

Apart from the changes in aspect code, changes in base code can be represented by the atomic changes for Java. We list the atomic changes defined for Java [16] in Table 2 to assist our analysis.

Abbreviation	Atomic Change Name
AF	Add a field
DF	Delete a Field
AM	Add an Empty Method
DM	Delete an Empty Method
CM	Change Body of Method
AC	Add an Empty Class
DC	Delete an Empty Class
LC	Change Virtual Method Lookup

Table 2: Atomic changes in base code

3.2 Inter-relationships between Atomic Changes

There are some inter-relationships between atomic changes that induce a partial ordering \prec on a set of them, with transitive closure \prec^* . That is, $C1 \prec^* C2$ denotes that $C1$ is a prerequisite for $C2$ ². This partial ordering indicates that when applying one atomic change to the program, all its dependent changes should also be applied in order to obtain a syntactically valid version. The benefit of defining such a partial ordering is that one can construct a semantic correctness intermediate version for the future analysis. We define two types of dependencies as follows.

3.2.1 Syntactic Dependence

Generally, syntactic dependence captures all prerequisite changes for one atomic change in order to form a valid version. The *syntactic dependencies* between atomic changes of base code has been

²To keep consistence, we use the same notions for the partial order definition as used in [16]

discussed in [15]. Here we just focus on the aspect-related constructs.

Adding / Deleting AspectJ Constructs. In our definition, all the adding atomic changes (**AA**, **INF**, **INM**, **AEA** and **ANP**) and deleting atomic changes (**DA**, **DIF**, **DIM**, **DEA** and **DPC**) represent adding or deleting an empty AspectJ construct. A new construct must be declared before using or making changes to its body, and similarly the construct body must be cleared before deleting the construct itself. Take the changes in Figure 2 for example, the new added `advice before():callPoints()` must be declared first before making changes in its body block, and the pointcut definition used in `advice before():callPoints()` must be added before declaring the advice, and all above changes must be applied after the addition of the empty aspect `InterceptAspect` construct. The syntactic dependencies between atomic changes involved in Figure 2 can be represented as:

```
AEA(before:callPoints()) < CAB(before:callPoints())
ANP(callPoints) < AEA(before():callPoints())
AA(InterceptAspect) < ANP(callPoints)
```

Similarly, the dependencies in deleting changes can be represented in a reverse ordering.

Changing AspectJ Constructs. In our model, only changes to the advice body, pointcut body, inter-type method body and inter-type field initializer have corresponding atomic changes (i.e., **CAB**, **CPB**, **CIMB** and **CIFI**). Other changes to the aspect constructs, such as *type changing of an inter-type field* or *renaming an inter-type method*, is decomposed into a delete change (**DIF**, **DIM**), an add change (**INF**, **INM**), and a corresponding modification change (**CIFI**, **CIMB** or **CAB**). In short, changing AspectJ constructs must follow the rule, that is, the constructs must be declared first before making any changes.

Class Hierarchy and Aspect Precedence Changes. Changes to the class hierarchy or aspect precedence declaration is related to the class or aspect definition it used. An aspect must be defined before it is used in any aspect precedence statement, but deleting any aspect precedence does not need any prerequisites. For example, assume that we add an aspect precedence declaration:

```
declare precedence IntroduceM2B: InterceptAspect
as an extra change in Figure 2, the dependence can be represented as:
```

```
AA(IntroduceM2B) < AAP(IntroduceM2B, InterceptAspect)
AA(InterceptAspect) < AAP(IntroduceM2B, InterceptAspect)
```

The dependencies in class hierarchy changes can also be handled in the same manner: the class or interface used in the declaration must be declared first.

Soften Exception Changes. Changes to the soften exception declaration is related to the pointcut definition it used. The constructs in pointcut must be first defined before used in any soften exception changes. For example, assume that we add a soften exception declaration:

```
declare soft: Exception: execution(A.p());
as an extra change in Figure 5, the dependence can be represented as:
```

```
AM(A.p()) < ASED(Exception: execution(A.p()))
```

3.2.2 Interaction Dependence

The dependencies we discussed so far are all syntactic dependencies in the aspect code, while the *interaction dependence* is the implicit inter-relationship involving both atomic changes in the as-

pect code and base code.

Inter-type Declaration Changes. As the syntactic dependence shows, the inter-type should be declared before any change to it. Take the changes in Figure 3 as an example, the field `A.countM` must be first introduced by aspect `IntroduceField` before it is used in method `C.m()`. Similarly, an introduced method can only be deleted when it is no longer referenced in base code. Hence, for the changes in Figure 3, we have the following dependencies related to the inter-type declaration change:

$$\begin{aligned} \text{AM}(\text{C.m}()) &\prec \text{CM}(\text{C.m}()) \\ \text{INF}(\text{A.countM}) &\prec \text{CM}(\text{C.m}()) \end{aligned}$$

Overloading Methods. When the inter-type method from aspect code overrides the existing one in the base code, it will change the method dispatch and cause an **LC** atomic change. This **LC** depends on the inter-type declaration changes in the aspect code. Consider the change in Figure 2, aspect `IntroduceM2B` introduces a method `m()` for class `B1`. For this change, an object of type `C` will no longer resolve to `A.m()`. This method dispatch change can be represented by **LC**. So the dependence between **LC** change and the inter-type declaration can be represented as:

$$\text{INM}(\text{B1.m}()) \prec \text{LC}$$

Abstract Aspects and Pointcuts. An abstract pointcut must be implemented in all the sub-aspects of an abstract aspect. Therefore, the declaration of an abstract pointcut must be deleted before the deletion of its implementation in the sub-aspect. This dependence relationship captures the way a new abstract pointcut is declared or deleted. Assume that program P defines an abstract aspect AA with a sub-aspect SA which extends AA , we add an abstract pointcut declaration of pCA into AA , and SA provides the implementation of pCA . Here, we have the dependence between atomic changes:

$$\text{ANP}(\text{AS.pCA}()) \prec \text{ANP}(\text{AA.pCA}())$$

Advice Invocation Changes. In AspectJ program, either changes like *renaming class members* in base code or changes like *adding new advice* in aspect code will cause the advice invocation behavior change. The advice invocation changes depend on the related source modifications. Consider the changes in Figure 4, the new added method `A.p()` matches the pointcut `callPoints` in aspect `InterceptAspect`. The corresponding **AIC** is represented as $\text{AIC} = \{ \langle \text{A.p}(), \text{before}():\text{callPoints}() \rangle \}$ in our model, and the dependence relationship can be represented as $\text{AM}(\text{A.p}()) \prec \text{AIC}$. Similarly, the changes in Figure 5 can be represented by **CPB**(`callPoints()`) and $\text{AIC} = \{ \langle \text{before}():\text{callPoints}, \text{A.m}() \rangle, \langle \text{before}():\text{callPoints}, \text{A.n}() \rangle \}$.

4. CHANGE IMPACT ANALYSIS MODEL

We next present a change impact analysis model for AspectJ programs. First, we construct the static call graph [14] for AspectJ program to determine all affected parts by traversing it from the modified node. Let P be an AspectJ program and $\text{Nodes}(P)$ and $\text{Edges}(P)$ be the node and edge sets in the call graph of P . We also assume there is a set of tests $T = t_1, \dots, t_n$, associated with the original program P . Each test driver t_i exercises a subset $\text{Nodes}(P, t_i)$ of P 's method. Likewise, $\text{Nodes}(P', t_i)$ from the call graph for t_i on the edited program P' .

To assist our analysis, we define the *Affected Node Set* $\text{ANS}(S)$ as all potential affected nodes in the call graph, where S is the modified node set. $\text{ANS}(S)$ represents all nodes reachable from any node $s \in S$, that is transitive closure of S in the graph. For example, in Figure 6, $\text{ANS}(\text{D}, \text{H}) = \{ \text{A}, \text{B}, \text{C}, \text{D}, \text{H} \}$.

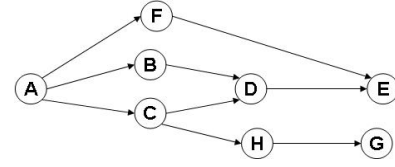


Figure 6: A sample call graph with $\text{ANS}(\text{D}, \text{H}) = \{ \text{A}, \text{B}, \text{C}, \text{D}, \text{H} \}$

4.1 The Affected Base Code

Either changes in base code or aspect code can affect the behavior of AspectJ programs. For those changes, the affected base code **AffectedBase1** due to changes of base code can be computed by the formalism defined in [16]. The affected parts of base code caused by changes in aspect code are:

$$\text{AffectedBase2} \equiv \text{ANS} \{ \text{AffectedBaseNodes} \}$$

$$\text{AffectedBaseNodes} \equiv \{ m \mid m \in \text{Nodes}(P') \wedge \langle m, a \rangle \in \text{AIC} \}$$

Thus, the total affected base code method **AffectedBase** is:

$$\text{AffectedBaseParts} \equiv \text{AffectedBase1} \cup \text{AffectedBase2}$$

4.2 The Affected Aspect Code

Changes in both base code or aspect code would also affect the behavior of an AspectJ program. In the AspectJ call graph, we treat the advice as a method-like unit represented by a node in the graph. By performing static analysis on the call graph, the affected parts of aspect code can be represented as:

$$\text{AffectedAspectParts} \equiv \text{ANS} \{ \text{AffectedAspectNodes} \}$$

$$\begin{aligned} \text{AffectedAspectNodes} \equiv \{ n \mid n \in \text{INM} \cup \text{DIM} \cup \text{CIMB} \cup \text{AEA} \\ \cup \text{CAB} \cup \text{DEA} \} \cup \{ m \mid \langle m, a \rangle \in \text{AIC} \} \end{aligned}$$

4.3 Solving Interactions between Base and Aspect Code

In an AspectJ program, an aspect can interact with a base class by four ways, that is through which the change impact occurs: (1) creating an object of the class from the aspect, (2) there is a call from a method or an advice in the aspect to a method of the class, (3) declaring a public introduction in the aspect to add a field, method or constructor to the class, and (4) weaving the code declared in advice of the aspect to the class code at join points. The first and the second ways are similar to class interactions in an object-oriented programs, which can be easily captured. In our model, change impacts caused by the third way can be represented by atomic changes **INF**, **DIF**, **CIFI**, **INM**, **DIM**, **CIMB** and **LC**. The changes impacts caused by the fourth way can be reflected by atomic changes **AEA**, **DEA**, **CAB**, **ANP**, **DPC**, **CPB** and **AIC**.

Therefore, changes caused by interactions between the base and aspect code can also be captured by the atomic changes, and all the affected parts in an AspectJ program can be represented as:

$$\text{AllAffectedNodes} \equiv \text{AffectedBaseParts} \cup \text{AffectedAspectParts}$$

4.4 Affected Tests and Affecting Changes

The affected test cases are the subset of existing test cases whose behavior may be affected by the changes. We define the affected tests **AffectedTests** as follows:

$$\text{AffectedTests} \equiv$$

$$\{ t_i \mid t_i \in T, \text{AllAffectedNodes} \cap \text{Nodes}(P', t_i) \neq \emptyset \}$$

For an affected test t_i , we define their corresponding affecting atomic

changes as below. Here A is the set of all atomic changes.

AffectingChanges \equiv

$$\{c' \mid c \in \text{Nodes}(P', t) \cap \text{AllAffectedNodes} \cap A, c' \preceq^* c\}$$

4.5 Change Impact Analysis: An Example

With respect to our original example in Figure 1 as well as the sequence of changes in Figures 2, 3, 4 and 5. The first change described in Figure 2 corresponds to the following atomic changes: $c_1 \equiv \text{AA}(\text{IntroduceM2B})$, $c_2 \equiv \text{INM}(\text{B1.m})$, $c_3 \equiv \text{INM}(\text{B2.m})$, $c_4 \equiv \text{CIMB}(\text{B1.m})$, $c_5 \equiv \text{CIMB}(\text{B2.m})$, $c_6 \equiv \text{AA}(\text{InterceptAspect})$, $c_7 \equiv \text{ANP}(\text{callPoints})$, $c_8 \equiv \text{AEA}(\text{before:callPoints})$, $c_9 \equiv \text{CAB}(\text{before:callPoints})$, $c_{10} \equiv \text{LC}(\text{B1.m})$, $c_{11} \equiv \text{LC}(\text{B2.m})$ ³, $c_{12} \equiv \text{AIC}(<\text{A.m}(), \text{before}():\text{callPoints}()>, <\text{A.n}(), \text{before}():\text{callPoints}()>)$. And we have: $c_1 \prec c_2 \prec c_4$, $c_1 \prec c_3 \prec c_5$, $c_6 \prec c_7 \prec c_8 \prec c_9$, $c_2 \prec c_{10}$, $c_3 \prec c_{11}$, and $c_8 \prec c_{12}$. For those changes, the affected methods are $\text{c.m}()$, $\text{B1.m}()$, $\text{B2.m}()$, $\text{A.m}()$ and $\text{A.n}()$, the affected tests are test1 , test2 , test3 and test4 .

The second edit in Figure 3 corresponds to the following changes: $c_{13} \equiv \text{AA}(\text{IntroduceField})$, $c_{14} \equiv \text{INF}(\text{A.countM})$, $c_{15} \equiv \text{INF}(\text{A.countN})$, $c_{16} \equiv \text{CIFI}(\text{A.countM})$, $c_{17} \equiv \text{CIFI}(\text{A.countN})$, $c_{18} \equiv \text{CM}(\text{C.m}())$, $c_{19} \equiv \text{CM}(\text{C.n}())$, $c_{20} \equiv \text{LC}(\text{C.m}())$, $c_{21} \equiv \text{LC}(\text{C.n}())$. Here, we have: $c_{13} \prec c_{14} \prec c_{16}$, $c_{13} \prec c_{15} \prec c_{17}$, $c_{14} \prec c_{18}$, $c_{15} \prec c_{19}$, $c_{18} \prec c_{20}$, and $c_{19} \prec c_{21}$. The affected methods are $\text{C.m}()$ and $\text{C.n}()$, and the affected test method is test4 .

The third change in Figure 4 corresponds to the following changes: $c_{22} \equiv \text{AM}(\text{C.p}())$ and $c_{23} \equiv \text{AIC}(<\text{before}():\text{callPoints}, \text{A.p}()>)$ and we have the relationship: $c_{22} \prec c_{23}$. In this change, no existing tests are affected and it may indicate that additional test cases should be written.

The last change in Figure 5 corresponds to the following atomic changes: $c_{24} \equiv \text{CPB}(\text{callPoints}())$, $c_{25} \equiv \text{AIC}(<\text{before}():\text{callPoints}(), \text{A.m}()>, <\text{before:callPoints}(), \text{A.n}()>)$ and we have the relationship: $c_7 \prec c_{24} \prec c_{25}$. In this change, the affected test method are test1 , test2 and test3 .

5. DEBUGGING SUPPORT FOR ASPECTJ PROGRAMS

In this section, we discuss the debugging support provided by our analysis approach as one of its applications. During software development process, the regression testing failure may indicate some potential defects in the current software version. At that time, programmers are often burden with the heavy tasks of searching through sources for those failure-inducing changes. Because the subset of editing which may affect a regression test can be only a small portion of the total source changes, it would be tedious for programmers to check each editing. Moreover, as we discussed in Section 1, changes in AspectJ programs may involve much more complex impact than the traditional programming languages. The failure-inducing changes may work together in a non-trivial manner and makes it extremely difficult to locate the exact reason of failure.

If regression tests fail, we start to debug AspectJ program by first computing the atomic changes and their inter-dependence relationships between two versions, then identify the affecting changes responsible for that test failure. Programmers may (repeatedly) select the suspected changes, and we construct valid intermediate program versions containing only the suspected changes and their prerequisites. Each of these intermediate versions can then be tested again using the failed regression test. According to the testing result, programmers can ignore certain changes that do not result in

³For simplicity, $\text{LC}(\text{B1.m}())$ is the method dispatch change caused by the introduction of $\text{B1.m}()$, and the same is $\text{LC}(\text{B2.m}())$.

the failure, and narrow down smaller set of changes until they locate the exactly failure reasons. In our Flota tool implementation, programmers can also rollback the current intermediate version to restore the original one, and begin exploration again.

We use the sample program in Figure 2 as an example to show the debugging support of our model. Associated with the original program in Figure 2 are four JUnit tests. These tests pass in the original program. However, after a sequence of source code modification in Figure 2, 3, 4 and 5, the test method Tests.test2 fails. As showed in Section 4.5, there are 4 affecting changes, c_2 , c_4 , c_{11} and c_{25} . The question is: *Which of those 4 changes are contributed for the test failure?*

In our approach, a programmer may first guess that the advice invocation change caused by aspect InterceptAspect may be responsible for the failure. Then he selects c_{12} and apply that to the original program. Then we automatically applies other necessary atomic changes c_1 and c_2 which c_{12} depends on to form a valid intermediate program version V_1 shown in Figure 7.

```
aspect InterceptAspect {
    pointcut callPoints(A a):call(* A.*(..))&&this(a);
    before(A a) callPoints(a) { a.i++; a.j++; }
}
```

Figure 7: Intermediate version V_1 after applying change c_{12}

Now, the programmer can execute Tests.test2 against V_1 to check whether it is passed or not. The programmer may find that it succeeds and then guess that c_{25} (*Change Pointcut Body*) may be the culprit of that failure, so he select and apply c_{25} to the intermediate version V_1 . Similarly, in our approach, we automatically applied the dependent changes c_{24} and c_7 and get another valid intermediate version v_2 shown in Figure 8 (here, we suppose programmer build the valid version upon the existing intermediate version.).

```
aspect InterceptAspect {
    pointcut callPoints(A a):call(* A.p(..))&&this(a);
    before(A a) callPoints(a) { a.i ++; a.j ++; }
}
```

Figure 8: Intermediate version V_2 after applying change c_{25}

Again, the programmer executes Tests.test2 against V_2 and find it passed. Now, he can ignore these applied changes and rollback to the original program version. The programmer continued to guess the responsible change may be c_4 and then apply that to the original version. After applying atomic change c_4 and its dependent change c_1 and c_2 , the intermediate version V_3 looks like as in Figure 9. After testing V_3 against Tests.test2 , programmer can find Tests.test2 fails again and the exact failure reason is c_4 .

```
aspect IntroduceM2B {
    public void B1.m() { i = i + 2; }
}
```

Figure 9: Intermediate version V_3 after applying change c_{25}

Through a toy example, the debugging support provided by our change impact analysis model can be useful to real world software (discussed in Section 7.7 and [23]), especially when there are hundreds or thousands of changes. In our approach, the syntactic dependence between each atomic change is calculated automatically and programmers only need to focus on the valid, interesting intermediate program versions and identify the failure-inducing changes by automating the iterative process of selecting suspected changes and applying them to the original version.

6. IMPLEMENTATION ISSUES

We next discuss some technical issues in the implementation of Celadon framework. We choose the most widely used and supported *ajc* AspectJ compiler [3] as the foundation of our implementation. The main technical issues involved in Celadon implementation are: (1) How can atomic changes be computed? and (2) How to construct AspectJ call graph to determine affected program parts?

6.1 Computing Atomic Changes

Our change impact model relies on the computation of atomic changes and their inter-dependence relationship. The AspectJ Development Tools (AJDT) [2] provides plenty of APIs for accessing and manipulating the abstract syntax trees (AST) of AspectJ program. The ASTs contain sufficient source information of AspectJ constructs and therefore ease the effort of pinpointing the locations of all these affecting changes. When comparing two program versions, we used a dynamic programming algorithm [21] to calculate the differences between two ASTs and then generate the corresponding atomic change set. Speaking in detail, inspired by the classic longest common subsequence (LCS) [19] problem, we first compute the common node pairs between ASTs, then remove those common parts from both versions temporarily. Therefore, the remaining nodes is either new added or deleted during changes. For the removed common node pairs, we continued to compute the common node pairs in their sub-AST nodes and do the same algorithm recursively. For the AIC change, we get the join point matching information when the *ajc* weaver composes the aspects and base code and then calculate the changes.

6.2 Determining Affected Parts

We have built call graph in [14] to assist change impact analysis for AspectJ programs. In our implementation, we use the RTA algorithm [7] to construct call graph of the base code. For the aspect code, we consider the advice as a method-like node with matching relationship represented by an edge from the join point. The complete call graph of AspectJ program is formed after the call graph of aspect code is *connected* into the base code call graph using the join point matching information. We treat the anonymous pointcut as a part of advice declaration, thus any changes to the anonymous pointcut will result in the definition change of associated advice. For the static initialization code block, we treat it as a special node in the call graph to model the related changes. Therefore, by traversing the call graph from the changed node, we can easily get all the affected method nodes. However, the dynamic pointcuts in AspectJ language poses a big challenge to precisely reflect the affected parts in static call graph analysis. As in AspectJ, the weaving process is quite complex for those dynamic pointcuts like *cflow*, *if* and *target*. Since a dynamic pointcut that statically matches a shadow could potentially not match that shadow at run time, it is quite difficult to accurately compute the impact of changing dynamic pointcuts. In our approach, when constructing the call graph for an AspectJ program, we conservatively assume that for all dynamic pointcuts, whether they match a shadow or not has to be determined at run time. Under this assumption, advices that are associated with dynamic pointcuts are connected to the corresponding call nodes, which means that changes in this dynamic advice will affect its calling method. Through this way, though approximately, we can safely calculate the affected methods concerning changes in a dynamic advice.

7. EMPIRICAL EVALUATION

To investigate the effectiveness and efficiency of our proposed technique, we have implemented Celadon: a change impact analysis framework for AspectJ programs. Celadon is designed as an Eclipse plugin. We have applied Celadon to perform change impact analysis on 24 versions of 8 AspectJ benchmarks collected from a variety of sources (Section 7.2). Our experimental results suggest that semantic differences between two program versions can be precisely captured by the atomic change representation (Section 7.4.1) and the impacted program parts as well as affected tests (Section 7.4.2, 7.4.3) can be determined with practical accuracy. The information provided by Celadon would be helpful for programmers to locate faults in AspectJ programs (Section 7.7). We will also discuss some issues of the experiment in Section 7.5 and 7.6.

7.1 Objectives

We investigate the following questions in the experiment:

- Can *atomic change* representation capture the semantic differences between two program versions accurately?
- Can impacted program parts, affected tests and their responsible affecting changes be determined effectively by our tool?
- What is the cost of the change impact analysis?

7.2 Subject Programs

Programs	#Loc	#Ver	#Me	#Shad	#Tests	%mc	%asc
Quicksort	111	3	18	15	27	100	100
Figure	147	4	23	5	20	100	100
Bean	199	3	12	8	15	100	100
Tracing	1059	4	44	32	15	100	100
NullCheck	2991	4	196	146	128	96.9	85.8
Lod	3075	2	220	1103	157	90.0	63.4
Dcm	3423	2	249	359	157	94.3	73.5
Spacewar	3053	2	288	369	132	88.5	74.0

Table 3: Subject Programs

We use eight AspectJ benchmarks shown in Table 3 for the experimental study. The first three and the spacewar example are included in the AspectJ compiler example package. The remaining programs were obtained from the abc benchmark package [1]. This group of benchmarks have also been widely used by other researchers to evaluate their work on test generation [20], performance measurement [9] and regression tests selection [10]. For each program, we made the first version v_1 a pure Java program by removing all aspectual constructs. For some program versions, we made additional modifications to produce more general changes rather than only changes within bodies of methods or advices. We also developed a test suite for each subject program.

Table 3 shows the number of lines of code in the original program (#Loc), the number of versions (#Ver), the number of methods (#Me), the number of shadows (#Shad), the size of the test suite (#Tests), the percentage of methods covered by the test suite (%mc), and the percentage of advice shadows covered by the test suite (%asc). *Advice shadow coverage* is defined as follows. An *Advice shadow interaction* occurs if a test executes an advice whose pointcut statically matches a shadow. The *Advice shadow coverage* is the ratio between *Advice shadow interactions* and the number of shadows in program.

7.3 Procedures

To assess how our approach helps change impact analysis, we take each pair of successive versions of AspectJ benchmark (i.e., v_1 and v_2 , v_2 and v_3 , etc) and its corresponding tests as the input of our tool. The tests are used in both versions. For each input,

change impact analysis is performed automatically by Celadon in the following steps:

First, it analyzes the source code of both program versions statically and decomposed the differences into a set of atomic changes together with their inter-dependence relationships.

Second, it generate call graphs for the benchmark program and tests.

Finally, change impact analysis is performed based on the output of the *First* and *Second* steps. The result is reported in terms of affected tests whose runtime behavior may have been altered by the applied changes. For each affected test, Celadon identifies a set of affecting changes that were responsible for the affected test case. For the updated program version, Celadon also determines the impacted program nodes in the call graph automatically.

7.4 Results

The experiments with Celadon were performed on 24 versions of 8 AspectJ benchmarks. Figure 10 to 17 shows the number of atomic changes between version pairs of each benchmark. The atomic changes between each version pairs are classified by the category, and the number varies greatly between 1 and 676. Section 7.4.1 explains more details about the result of atomic changes in our experiments. Table 4 summarizes the affected tests and affecting changes for each version of those benchmarks, and Table 5 shows the affected method nodes in call graph of the analyzed programs. Note that not all atomic changes occur in each benchmark. In Table 4 and 5, each AspectJ program version is labeled with its number - eg. Q2 corresponds to version v_2 of Quicksort, N4 is version v_4 of Nullcheck, etc.

7.4.1 Atomic Changes

Quicksort: Figure 10 shows the number of atomic changes between each pair of versions in the Quicksort benchmark. The height of each bar indicates the number of corresponding kind of atomic changes. There are totally 10 atomic change categories derived from the differences between each version pair. As seen in the diagram, the most frequent changes in Quicksort are **AIC** (*Advice Invocation Change*) and **CAB** (*Change Advice Body*), each has 8 changes between version v_2 and v_3 . The second frequent change is **CPB** (*Change Pointcut Body*), which account for 7 between v_2 and v_3 . Two atomic changes in Figure 10 does not occur between version v_1 and v_2 (the v_2 bar), namely **DEA** (*Delete Empty Advice*) and **DPC** (*Delete Pointcut*), because v_1 is a pure Java program which does not have any aspect constructs. Between version v_2 and v_3 , there is no **AA** (*Add Aspect*) change, because version v_2 and v_3 both has the same aspect `Stats`. However, version v_2 and v_3 have different pointcut and advice to implement the crosscutting concerns in Quicksort. Thus it causes a number of related changes like **AEA** (*Add Empty Advice*), **DEA** (*Delete Empty Advice*), **ANP** (*Add New Pointcut*), etc.

Figure: Figure 11 shows the number of atomic changes between each version pair of the Figure benchmark. From the diagram, we can see most of the changes are in low number except for common Java changes like **CM**, **AM** and **DM**. Version v_2 added an aspect `DisplayUpdating` which contains a pointcut `move()` together with an advice after returning: `move()`. Therefore, the v_2 bar represent these changes by one **AA** (*Add Aspect*), one **ANP** (*Add New Pointcut*), one **CPB** (*Change Pointcut Body*), one **AEA** (*Add Empty Advice*), one **CAB** (*Change Advice Body*) and one **AIC** (*Advice Invocation Change*). Changes between version v_2 and v_3 and changes between v_3 and v_4 are mainly Java construct changes. The aspect related atomic changes are mainly due to the editing like *adding* or *deleting* pointcut designator.

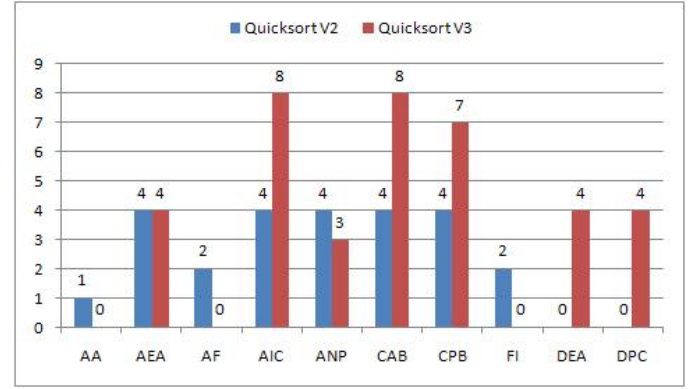


Figure 10: Number of atomic changes between each version pair of Quicksort benchmark

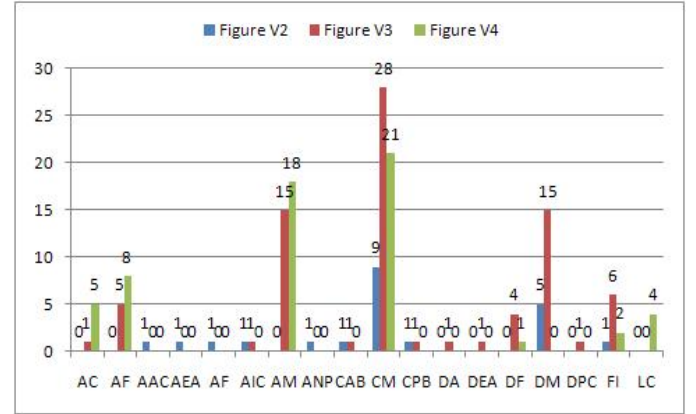


Figure 11: Number of atomic changes between each version pair of Figure benchmark

Bean: Figure 12 shows the number of atomic changes between each version pair of the Bean benchmark. Between the successive versions, there are 18 categories of different changes in which 13 belongs to the aspect construct changes defined in Table 1. Besides the **CM** change, version v_2 declares 5 inter-type methods (eg. `method Point.addPropertyChangeListener`, `Point.hasListeners`, etc) into the original version v_1 , which correspond to the 5 **INM** (*Introduced New Method*) together with 5 **CINM** (*Change Introduced Method Body*) changes as shown in the v_2 bar. Version v_2 added declaration `declare parents: Point implements Serializable` in the `PointBound` aspect. Therefore, there is corresponding **AHD** (*Add Hierarchy Declaration*) changes in v_2 bar to capture this change. The differences between v_2 and v_3 mainly comes from the different implementation of pointcut and advice in the `BoundPoint` aspect, and these changes are reflected by 3 **AIC**, 2 **AEA** and 3 **CAB** changes.

Tracing: The result of Tracing benchmark is shown in Figure 13. There are total 15 categories of atomic changes among 3 version pairs, in which 8 are aspect-related changes. The most frequent changes of aspect feature is **AIC** (*Advice Invocation Change*), while the overall most frequent change is the **CM**. Between version v_1 and v_2 , *adding empty advice* (captured by 4 **AEA** change) and *adding empty method* (captured by 8 **AM** change) result in four **AIC** changes, which indicate the addition of advice, join point matching pairs. The changes between version v_2 and v_3 (repre-

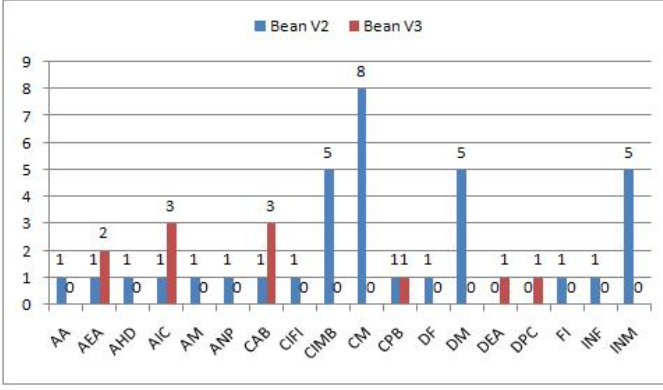


Figure 12: Number of atomic changes between version pair of Bean benchmark

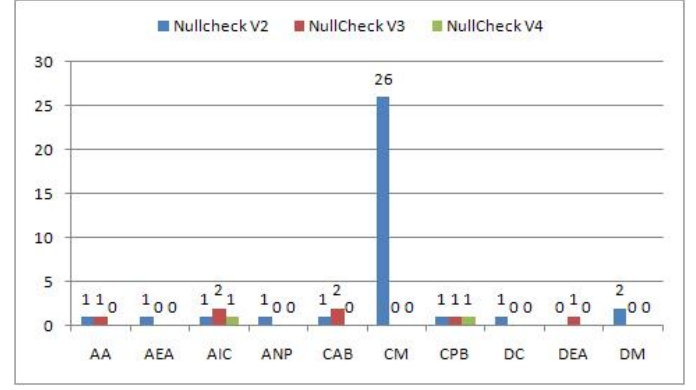


Figure 14: Number of atomic changes between each version pair of Nullcheck benchmark

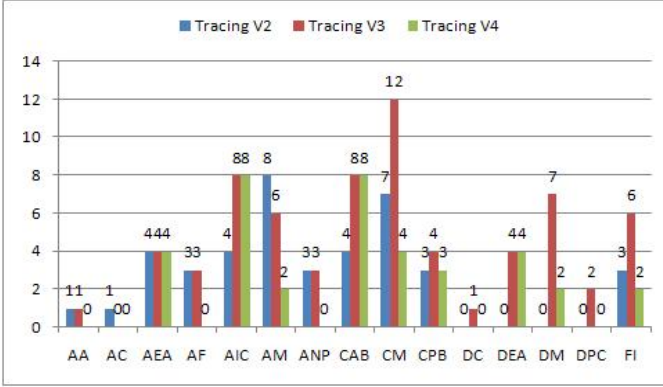


Figure 13: Number of atomic changes between each version pair of Tracing benchmark

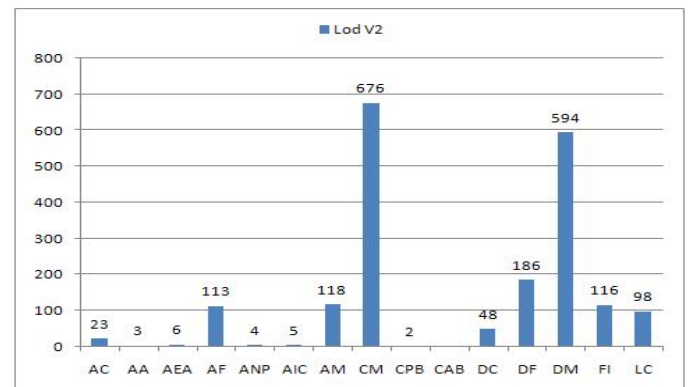


Figure 15: Number of atomic changes between each version pair of LoD benchmark

sented by bar v_3) contains 4 **DEA** (*Delete Empty Advice*) and 4 **AEA** (*Add Empty Advice*) changes, which represents the fact that v_3 uses different advices to implement the crosscutting concerns⁴.

NullCheck: Though Nullcheck is a moderate-sized AspectJ application, which has 1059 lines of code in the original version, there is not many changes between its successive versions. However, the **CM** change is surprisingly higher than others between v_1 and v_2 (represented by bar v_2), which indicates there is an intensive editing of *changing the existing method body*. There are also few changes in other version pairs. The source editing of aspect-related feature is mainly in package `lib.aspects.codingstandards`. Version v_3 changes the advice declaration `after() returning(Object lRetVal)` to `Object around(): methodsThatReturnObjects()`, which is captured by 1 **DEA**, 2 **CAB** and 1 **AEA** change. Interestingly, there is few changes reported by Celadon between version v_3 and v_4 . The only two atomic changes reported in version v_4 is the change of `pointcut methodsThatReturnObjects()` in aspect `EnforceCodingStandards`, which is capture by a **CPB** (*Change Pointcut Body*) change. This change of pointcut body causes the existing advice matches less join points than before, therefore, one **AIC** (*Advice Invocation Change*) is reported to reflect this semantic difference.

LoD: There are only two versions of LoD benchmark available

⁴ Actually after inspecting the source differences, we found version v_3 moves (first delete then add) the original advice in aspect `TraceMyClasses` to a new abstract aspect `Trace` to implement the crosscutting concerns

from the `ajbenchmark` [1] suite, and the atomic changes between version v_1 and v_2 is shown in Figure 15. The number of changes varies greatly between 2 to 676 between these two versions. The reason why there are surprisingly so many atomic changes is that version v_1 and v_2 have fundamentally different package organization of source code. Version v_1 has two top level packages `weka` and `lawOfDemeter` under the `src` directory. However, Version v_2 removed the `weka` package and added two new packages `jsim` and `certrevsim`. Therefore, the source code changes under package `weka`, `jsim` and `certrevsim` are all regarded as differences between version v_1 and v_2 . Changes under these three packages is captured by 23 **AC**, 113 **AF**, 118 **AM**, 676 **CM**, 98 **DM** and the corresponding deleting atomic changes. The addition of three aspect `Check`, `Percflow`, and `Pertarget` under package `lawOfDemeter` is captured by 3 **AA** (*Add Aspect*), 4 **ANP** (*Add New Pointcut*) together with the **CPB** change and 6 **AEA** (*Add Empty Advice*) together with the corresponding **CAB** (*Change Advice Body*) changes.

Dcm: Dcm is the largest benchmark among the subjective programs. Its experiment result is shown in Figure 16. Unlike the intensive source editing in LoD, there are not many changes between version pairs of Dcm. Version v_2 added a new package `DCM` to the source directory, in which are mainly Java code changes. Four new added aspects `Pointcuts`, `ClassRelationship`, `AllocFree` and `Metrics` in package `DCM`, `DCM.handleGC` and `DCM.handleMetrics` as well as their members change are captured by 4 **AA**, 8 **AEA**, 1 **AHD**, 8 **AIC**, 3 **ANP**, 8 **CAB**, 1 **CIMB**, 1 **INM**, and 3 **CPB**

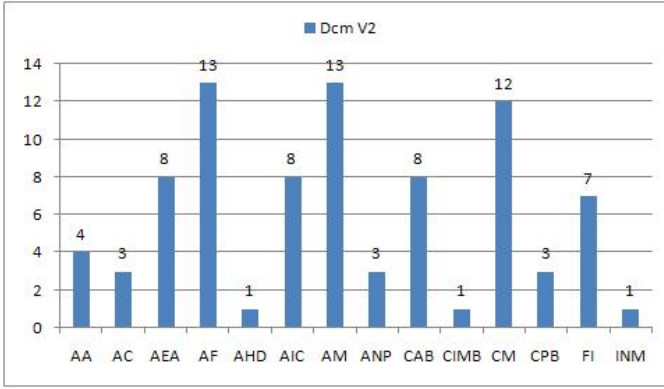


Figure 16: Number of atomic changes between each version pair of Dcm benchmark

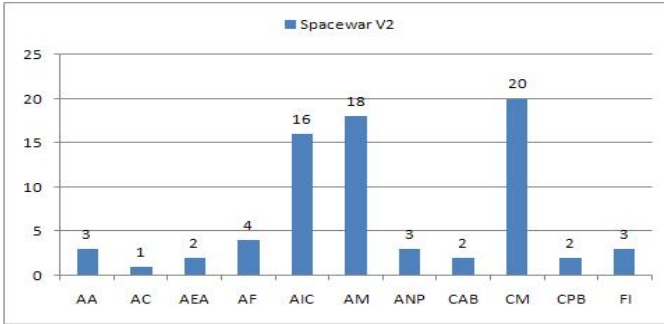


Figure 17: Number of atomic changes between each version pair of Spacewar benchmark

changes.

Spacewar: The result of Spacewar benchmark was shown in Figure 17. Among the 11 atomic change categories, six was aspect-related changes defined in Table 1. Besides common Java atomic changes **CM** and **AM**, the most frequent changes is the **AIC** (*Advice Invocation Change*). These **AIC** changes may indicate that the new added aspect may significantly affect the runtime behavior of the original program. For example, the new added aspect `RegistrationProtection` in the `Registry.java` file, which intercept the calling to method `Registry.regiser(SpaceObject)` and `Registry.unregiser(SpaceObject)`, is used to check the designed invariant during the runtime.

7.4.2 Affected Tests and Affecting Changes

Table 4 shows the affected tests and their affecting changes for each benchmark version pair. The number in each column represents the number of total atomic changes (Total Number), percentage of affected tests (% at) and percentage of atomic changes (% ac) that should be responsible for affected tests. On average, 74.2% of the tests are affected and 69.2% of the atomic changes are responsible in each version pair. Interestingly, there were several version pairs over which all tests were affected. For example, all tests were affected between version v_1 and v_2 (see the row T_2 in Table 4), despite the fact that there were only 41 atomic changes during this time. The reason is that version T_2 defines two pointcuts `myConstructor():myClass() && execution(new(...))` and `myMethod():myClass() && execution(* * (...))` in aspect `TraceMyClasses`, which crosscut all constructors and methods that are always executed at run time. In the row of T_3 and T_4 ,

Celadon also reported 100% tests affected. Because T_3 refactors the `TraceMyClasses` aspect in T_2 into an abstract aspect `Trace` and a concrete aspect `TraceMyClasses`, while T_4 add an argument (Object obj) in each pointcut declaration of version T_3 , which leads to potential behavior changes of each method execution. Those changes in Tracing benchmark indicate that in AspectJ programs, even small changes (such as *Adding New Pointcut* or *Change Pointcut Body*) would dramatically change the program structure and affected program behavior significantly. Therefore, in principle, a change impact analysis tool could inform the user that all tests should be rerun to validate the changes when an observation of this kind is found.

The number of *Affecting Changes Percentage* column indicates the total responsible changes of all affected tests. For some versions, the affecting atomic changes percentage reported by Celadon is reduced to 16.9% (F_4), 36.5% (T_2) and 47.8% (T_3). This means that our approach has the potential application of isolating the responsible changes if regression tests fails.

Version	Total Number	% at	% ac
Q2	24	100%	66.6%
Q3	38	100%	71.2%
F2	22	60%	54.5%
F3	80	80%	57.5%
F4	59	30%	16.9%
B2	35	80.0%	85.7%
B3	11	40.0%	100%
T2	41	100%	36.5%
T3	69	100%	47.8%
T4	37	100%	72.9%
N2	35	78.1%	88.5%
N3	7	78.1%	85.7%
N4	2	50.7%	100%
L2	1979	100%	75.4%
D2	85	86.2%	67.1%
S2	74	30.3%	85.13%

Table 4: Total atomic change number (Total Number), percentage of affected tests (% at) and percentage of affecting atomic changes (% ac)

7.4.3 Affected Nodes in Call Graph

As shown in Table 5, the percentage of affected nodes (% Affected Nodes) in call graph ranges from 10.6% to 95.6%. The data shown in row Q_2 , N_2 and N_3 indicates that both *adding aspect features* into the original object-oriented systems and *changing the existing aspect features* would widely affect the whole programs. In row N_2 and N_3 , nearly all of the nodes in call graph are affected, because N_2 adds a pointcut `methodsThatReturnObjects(): execution(Object+ *.* (...))` to cross cut base Java methods and N_3 changes the advice declaration `after() returning(Object lRetVal): methodsThatReturnObjects()` which are always executed at runtime. Celadon reports such potentially affected methods during the change impact analysis, and this information informs developers sufficient tests should be developed to cover all these impacted methods.

7.5 Threads to Validity

Like any empirical evaluation, this study also has limitations which must be considered. Although we applied Celadon on 24 versions of 8 AspectJ benchmarks, we expect that Celadon can be applied to a wide range of AspectJ programs. In our experience, we performed change impact analysis to only subject programs listed in Table 3. Though these subject programs are well know examples and the last three ones are among the largest programs that we

Version	Nodes Num	Affected Nodes	% Affected Nodes
Q2	22	12	54.5%
Q3	23	13	56.5%
F2	26	5	19.2%
F3	32	17	53.1%
F4	74	24	32.4%
B2	73	24	32.8%
B3	45	14	31.1%
T2	112	22	19.6%
T3	112	22	19.6%
T4	118	12	10.6%
N2	708	677	95.6%
N3	709	683	96.3%
N4	709	126	17.7%
L2	759	705	92.8%
D2	851	382	44.8%
S2	1162	446	38.4%

Table 5: Total method nodes in call graph (Nodes Num), number of affected method nodes (Affected Nodes) and percentage of affected nodes (% Affected Nodes)

could find, they are smaller than traditional Java software systems. For this case, we can not claim that these experiment results can be necessarily generalized to other programs.

On the other hand, threats to internal validity maybe mostly lie with possible errors in our tool implementation and the measure of experiment result. To reduce this kind of threats, we performed several careful checks. For each result produced by Celadon, we manually inspect the corresponding code to ensure the correctness.

7.6 Analysis Cost

The change impact analysis performed by Celadon runs in practical time. For example, the experiment was conducted on a DELL C521 PC with AMD Sempron 3.0G HZ CPU and 1.0GM memory, for the two largest programs `LoD` and `Dcm`, calculating atomic changes from two successive versions takes, on average, approximately 8.3 and 9.2 seconds, respectively. Computing the set the affected tests for each version pair takes about 1.4 and 1.8 seconds, and computing affecting changes takes on average approximately 2.0 and 2.6 seconds, respectively. Determining the impacted program parts in the new version costs about 0.9 and 1.1 seconds on average, respectively.

7.7 Discussion

Next, we discuss an application of our change impact analysis technique, locating bugs in AspectJ programs using our Flota tool. As shown in Section 7.4.2, the number of atomic changes reflect the source modifications between program versions and even small changes in AspectJ program can dramatically affect the behaviors of original program. Therefore, if a regression test fails after an editing session, it may be tedious to find out the exactly reasons by manually inspecting all source modifications, especially when having hundreds or thousands of changes in large systems. As discussed in Section 5 and 7.4.2, our change impact analysis has the potential application of isolating the responsible changes and providing debugging support for AspectJ programs. We have implemented the debugging support technique in our fault localization tool - Flota which is built on top of Celadon. Flota can effectively reduce the number of the responsible changes when a specific test fails, by repeatedly constructing the intermediate program version. For example, in our experimental study, one test `testRevocationInfo` in package `certrevsim` in the 2nd version of `Dcm` benchmark fails, Celadon reports a total 36 affecting changes, which accounting 42.5% of the total changes. When start-

ing debugging, programmers may suspect one change: `CAB(Dcm.handleMetrics.Metrics.after())` may be responsible for the failure, and he apply it to the original version. Flota automatically constructs a valid intermediate version which contains only the applied change and its prerequisites (reduced to 12 changes which account for 14.2% of the total number). Then programmer can test this program version again to see passed or not (and then decide to roll back or narrow down). Through this way, Flota can help programmer fast narrow down the exactly responsible changes. Since this paper mainly focus on the change impact analysis of AspectJ programs, more information about fault locating techniques and Flota implementation can be found in [23].

8. RELATED WORK

We discuss related work in the areas of change impact analysis, regression testing, and delta debugging.

8.1 Change Impact Analysis

Many change impact analysis techniques [5, 6, 13, 15, 16] have been proposed in recent years, which are mainly focused on procedural or object-oriented languages. Ryder et al. [16] use the atomic changes to determine the effects of a set of source code changes for Java, and a more sophisticated definition of the dependencies between atomic changes was given in [8] as well as the corresponding analysis tool [15]. Our work is an extension of the concept of atomic changes to aspect-related constructs to perform the change impact analysis for AspectJ programs.

Zhao [24] presents an approach to supporting change impact analysis of aspect-oriented software based on program slicing. Storer and Graf [18] focus on the semantic modification of base code and introduce a delta analysis to deal with the *fragile pointcut* problem, based on a comparison of the sets of matched join points for two program versions. Shinomi and Tamai [17] discuss the impact analysis of aspect weaving and its propagation through the base program and aspects. They focus on the change impact caused by the weaving aspect. In our work, we present the atomic changes for aspect-related constructs to capture program semantic changes in the source code level. The change impact model can be used to determine the affected program parts, affected tests and their responsible changes effectively.

8.2 Regression Test Selection for AspectJ programs

Change impact analysis is also related to the regression test selection techniques for aspect-oriented programs [10, 25]. The regression tests selection techniques aim at reusing test cases from an existing test suite to retest the new version of programs in order to reduce the testing effort. Our approach can also be used to identify the affected test cases in AspectJ programs and help handle the regression selection problems that are unique to aspect-related constructs.

8.3 Delta Debugging

Delta debugging, first proposed by Zeller [22], is used to identify the reasons for a program failure among large sets of textual changes. It searches the entire set of changes and builds the intermediate programs by repeatedly applying different subsets of the changes to the original program. Zeller's approach only used textual differences (Changes like changing one line or one character) between succeeding and failing program executions. However, the debugging support provided by our model taken into account the dependence relationships between atomic changes to ensure compellability. In our approach, programmers choose the most likely

failure-inducing changes from the atomic change set to generate meaningful intermediate program version. And they need not to consider the syntactical dependence between atomic changes.

Crisp [8] is a debugging tool for Java, which uses a similar approach as our model provided to construct the intermediate version for Java programs. In our work, we focus on the aspect-oriented programs and present a catalog of atomic changes in AspectJ programs, which can be used to provide valuable debugging support.

9. CONCLUDING REMARKS

In this paper we presented a change impact analysis approach for AspectJ programs. Our analysis model uses an *atomic changes* representation to capture precisely the semantic changes for AspectJ programs during software evolution. We construct static AspectJ call graph to effectively identify the impacted program fragments, affected tests and their responsible affecting changes. As one of the applications, we also present the debugging support for AspectJ programs provided by our model. Our approach is based on the static analysis of the source code of AspectJ programs, therefore it abstract away the low-level details that are specific to a compiler implementation. Our empirical study indicates the proposed technique can provides useful tool supports and valuable information during the evolution of AspectJ software.

In our future work, we will improve our current change impact model to capture the semantic changes more precisely, especially the dynamic pointcut changes in AspectJ programs. We also will extend our model to support automatic debugging for AspectJ programs and investigate more applications of this analysis approach.

Acknowledgements

This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 60673120), and Shanghai Pujiang Program (Grant No. 07pj14058). We would like to thank Chong Shu and Si Huang for their efforts on this work.

10. REFERENCES

- [1] The AspectBench Compiler.
<http://abc.comlab.ox.ac.uk/>.
- [2] AspectJ Development Tools (AJDT).
<http://www.eclipse.org/ajdt/>.
- [3] The AspectJ Team. The AspectJ Programming Guide. Online manual, 2003.
- [4] Junit, Testing Resources for Extreme Programming, 2006.
- [5] T. Apiwattanapong, A. Orso, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–137.
- [6] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.
- [7] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *In OOPSLA '96 Conference Proceedings, San Jose, CA, October 1996*, pages 324–341, 2004.
- [8] O. Chesley, X. Ren, and B. G. Ryder. Crisp: A debugging tool for Java programs. In *Proc. International Conference on Software Maintenance (ICSM'2005)*, Budapest, Hungary, September 27–29, 2005.
- [9] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the dynamic behaviour of AspectJ programs, 2004.
- [10] R. A. Guoqing Xu. Regression test selection for AspectJ software. In *In Proc. of the 29th International Conference on Software Engineering (ICSE07)*, pages 65–74, May 2007.
- [11] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proc. 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, 2004.
- [12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, pages 220–242, 1997.
- [13] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proc. 25th International Conference on Software Engineering*, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.
- [14] N. Li. The call graph construction for aspect-oriented programs. Master's thesis, School of Software, Shanghai Jiao Tong University, March 2007 (in Chinese).
- [15] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.
- [16] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *PASTE'01*, pages 46–53, 2001.
- [17] H. Shinomi and T. Tamai. Impact analysis of weaving in aspect-oriented programming. In *Proc. 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 657–660, 2005.
- [18] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proc. 21st IEEE International Conference on Software Maintenance*, pages 653–656, Washington, DC, USA, 2005. IEEE Computer Society.
- [19] R. L. R. Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithm, 2nd Edition*. MIT Press, 1996.
- [20] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 190–201, New York, NY, USA, 2006. ACM Press.
- [21] W. Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991.
- [22] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proc. 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.
- [23] S. Zhang and J. Zhao. Locating faults in AspectJ programs. Technical Report SJTU-CSE-TR-07-04, Center for Software Engineering, SJTU, July 2007.
- [24] J. Zhao. Change impact analysis for aspect-oriented software evolution. In *Proc. 5th International Workshop on Principles of Software Evolution*, pages 108–112, May 2002.
- [25] J. Zhao, T. Xie, and N. Li. Towards regression test selection for AspectJ programs. In *Proc. 2nd workshop on Testing aspect-oriented programs*, pages 21–26, July 2006.